Gouvernement du Canada
Ministère des Communications

Government of Canada
Department of Communications

**Le Centre canadien de recherche sur l'informatisation du travail**
**Canadian Workplace Automation Research Centre**

Canada

92-05728/

# THE TULIP PROJECT

by

**Ron Ferguson**

This report is the result of research done at Canadian Workplace Automation Research Centre under the direction of Joel Muzard, Experts Systems Group

Department of Communications

Laval

November 1989

Nº de cat. Co 28-1/40-1989F
ISBN 0-662-95734-2

Les opinions émises dans ce rapport n'engagent que l'auteur.

* This report is also available in English

## Table of contents


Introduction ..... 2

Tulip Implementation ..... 2

Building Tulip ..... 4

The Tulip Source Files ..... 6

Heuristics for Business Card Parsing ..... 12

The Architecture of Tulip ..... 12

Tulip's Knowledge Representation ..... 14

Skolem Functions ..... 14

Binary Predicates ..... 16

Know and Inferred Facts ..... 18

Representation of Verbs ..... 19

**REFERENCES** ..... 21

## Introduction

This report discusses the implementation of Tulip on Sun workstations. The capabilities and limitations of Tulip are described in the Sun Tulip User Manual [Ferguson 89b].

## Tulip Implementation

The Sun version of Tulip is written in Quintus Prolog plus a few C routines. It runs under the UNIX operating system. The source code is maintained using UNIX SCCS (Source Code Control System). The source code for Tulip is contained underneath the directory '/home/condor/ferguson/tulip'. All (sub)directories mentioned in this report will be sub-directories mentioned in this report will be sub-directories of '/home/condor/ferguson/tulip'. The 'tulip' directory contains the following subdirectories :

| | |
|---|---|
| **main** | - contains the main source code for parsing sentences. |
| **business-card** | - contains code for parsing business cards and names. |
| **expert-system** | - contains code for 'built-in'knowledge of dates and times, arithmetic operations, and answering questions. |
| **extraposition-grammar** | - code for translating extraposition grammars to Prolog clauses. |
| **foreign-files** | - code for interfacing with C routines. |
| **help** | - Tulip's help facility. |
| **knowledge** | - knowledge files which can be read by Tulip. |
| **lexicon** | - Tulip's dictionary. |

| | |
|---|---|
| **transform** | - shell for writing Prolog preprocessors. |
| **tulip** | - executable images. |
| **documentation** | - documentation of Tulip. |

Building Tulip

In subdirectory 'main', the files parse.pl and date-time-parse.pl contain extraposition grammars [Pereira 83] which must be translated into Prolog. The corresponding Prolog files are translation-of-parse.pl and translation-of-date-time-parse.pl, respectively. If one of the files containing an extraposition grammar is changed, it must be retranslated. To retranslate a file, load Tulip from 'main' and compile xg-fix.pl :

```
compile('../extraposition-grammar/xg-fix').
```

Then run 'gxg' (defined in xg-fix.pl) :

```
gxg.
```

gxg will prompt for an input file and an output file. Enter parse.pl (or date-time-parse.pl) as the input file and translation-of-parse.pl (or translation-of-date-time-parse.pl) as the output file.

In subdirectory 'business-card', the file address.pl contains an extended Definite Clause Grammar for parsing business card [Ferguson 89]. This file must be preprocessed if it is changed. To run the preprocessor, load Tulip from 'main'and run 'gopt':

```
gopt.
```

gopt will prompt for an input file and an output file. Enter ../business-card/address.pl as the input file and ../business-card/tr-address.pl as the output file.

The file 'tulip-system.pl'in 'main'contains commands to compile the Tulip system. If one compiles 'tulip-system.pl', Tulip is recompiled and an image file is saved in the 'tulip'sub-directory. The 'main'sub-directory contains an executable file called 'tulip'which loads this image.

Tulip uses a large lexicon. The lexicon is expressed as Prolog facats in a series of files. When Tulip was first ported to the Sun, these lexicon files were loaded as part of Tulip. However, the response time Tulip on Sun

3/50 machines was found to be poor. It seemed likely that the poor response time was partly due to the memory paging that was required because of the size of the Tulip program. In an attempt to increase the speed of Tulip, the lexicon was stored in dbm files and only the lexicon entries for words which actually appear in an input sentence were read into memory. Off-loading the lexicon in this manner reduced the amount of virtual memory required by the Tulip program, and as a result Tulip now runs 3.4 times faster.

The database lexicon consits of records. Each record has a key field which is an English word and a content field which is an integer. The integer is a bit vector representation of a set. The set is a subset of lex_set which is defined in lexical-dictionary-interface.pl. The elements of this set are the names of Prolog lexicon predicates which take an English word as their only argument. For example, if 'noun(dog)'is a fact in the Prolog lexicon, then 'noun'will be a member of the set associated with the word 'dog'.

The database lexicon is contained in the files 'lexicon.pag'and 'lexicon.dir'in the directory 'lexicon/dbm/'. If these files have to be recreated, the following procedure can be followed :

Delete the files 'lexicon.pag'and 'lexicon.dir'.
Create empty 'lexicon.pag'and 'lexicon.dir'files.
Go to the 'main'sub-directory.
Compile 'lexicon-system.pl'(this is a compile file for the Prolog lexicon).
Load but do not enter Tulip (remain in Quintus Prolog).
Enter 'create_lexicon.'

Note that it takes several hours to make the database lexicon from scratch. If one only wants to make minor changes to the database lexicon, it may be more convenient to modify the existing database lexicon using the predicates in 'dbm.pl'and 'fixed-set.pl'which are located in the 'lexicon/dbm'and 'main'directories, respectively. Of course, if one does modify the database lexicon directly, one should make corresponding changes to the Prolog lexicon files. See the file 'main/lexicon-system.pl'for a list of the Prolog lexicon files.

A description of the lexicon files is given in [Robb 88].

Some of the lexicon files contain predicates with an arity greater than one. These predicates are not contained int he database lexicon. In order to reduce the paging load, the 'tulip-system.pl'file will include these files in Tulip only if the fact, 'lots_of_memory'ocurs in the Prolog database. Otherwise, these files are not included when Tulip is compiled.

The Tulip Source Files

The following table gives the names of the non-lexicon source files in the Tulip system and a description of the contents of each file. The pathnames given for the files are with respect to the 'main'sub-directory. Each of these files has '.pl'as a file extension. The file extension is not shown in the table.

parse
: Main English Grammar

date-time-parse
: Date and Time Parsing

../business-card/address
: Parser for Business Card

utilities
: Generally Useful Predicates

collections
: Operations on Sets

../help/help
: Help Menu System

tulip-date
: Tulip date-includes day of the week

accept-sentence
: Read a sentence.

../transform/transform
: Shell for transforming Prolog Clauses (Preprocessors)

tokenize
: Tokenizer : string = list of tokens (reversible)

lexical-structure
: Lexical Structure Data Type Definition

guess-lexical-type
: Guess the lexical categories of words which are not in the lexicon

../extraposition-grammar/xg-fix-aux
: Basic Nonterminald for Extraposition Grammars/Record of parsed words

fp
: Operations on arrays (lists of lists)

abbrev
: Abbreviations

synonym
: Synonymous Words

french-english
: English Translation of French Words

new-lexicon
: Basic Lexicon

attachment-control
: Masks which control modifier attachment.
Ref. pg. 64, "Logic for Natural Language Analysis", F. Pereira

top
: Top Level Prolog routines for Tulip.

execute command
: Process Tulip's built-in commanda

chat-lexicon-interface
: Lookup words in the lexicon

morphology
: Morphological Analysis

slot-filler
: Replace words int he parse tree with their 'meanings'.

simplify
: Simplify 'quant'trees by removing 'true'leaves

scope-determination
: Determine the scope of English quantifiers.

template
: Interface to templates which define the 'meaning'of words.

predicate-calculus
: Convert Predicate Calculus formulae to clausal form.

modify-determiner
: Change the determiners in the parse trees of assertions so that the quantification is more explicitly represented.

knowledge-file-access
: Read from and write to knowledge files.

transform-answer
: Add facts to or query the knowledgebase

parse-names
: Parse a name, address or other non-grammatical entity.

anaphora
: Handle anaphoric references

triggers
: Trigger actions based on assertions being made to the knowledgebase.

parse-mods
: Modification to grammar to handle conjunction of verb phrases

special-nps
: Special parsers for identifying 'named'objects (eg. person and company names, addresses, phone numbers).

../business-card/gr-transform
: Translate an extended DCG grammar (for business cards) to Prolog clauses.

../business-card/address-util
: Token analysis for business cards

../business-card heuristics
: Heuristics to label those parts of a business card which the business card grammar was unable to parse.

../business-card/semantics
: Process the specification of relationships between business card objects

infer
: Basic inferences

loop-checking
: Check rules for recursive loops.

database-access
: Removable clause assertion

significant-words
: Assertion retrieval via word matching

date-time-facts
: Facts about dates and times

mrl_temp
: Interface to Meaning Representation templates for verbs

type-hierarchy
: Specification of built-in type hierarchy for template matching.

../expert-system/date-time-expert
: This is the Date-Time Expert System Engine i.e. calculator

../expert-system/talkr
Query evaluation

../expert-system/aggreg
Execute arithmetic operations

quintus-kludges
Predicates which can not (or have not) been properly defined in the Quintus version of Tulip.

The preceding table was created by a Prolog program in 'documentation/file-comments.pl'which gets the header comment from each file. The 'main/tulip-system.pl'file contains (commented out) commands to execute the header comment program. The file 'main/header-comment.pl'contains a template for header comments.

## Heuristics for Business Card Parsing

As described in [Ferguson 89], heuristics were used to classify those parts of a business card grammar. However, Tulip's business card grammar may be applied to the parsing of information for which the business card heuristics are not appropriate (ie. organizations which are engated in AI). Therefore, Tulip has been modified so that the heuristics are only applied if the fact 'apply_heuristics'is defined. The 'apply_heuristics' flag is checked in 'business-card/address.pl'.

## The Architecture of Tulip

The architecture of Tulip is based on that of Chat-80 [Pereira 83]. Differences between Tulip and Chat-80 are discussed in [Ferguson 88a].

Tulip uses a phased architecture to process sentences. The entire sentence is analysed by one phase before it is passed to the next phase. The phases involved in the parsing of an English sentence are :

| Process | Predicate |
|---|---|
| read input | read_lines/6 |
| tokenization | tokenize_parse/3 |
| syntactic lexicon lookup | lexical_lookup/2 |
| finding significant words | significant_words/4 |
| syntactic parsing | sentence/6 |
| semantic lexicon lookup | i_sentence/2 |
| quantifier scope determination | clausify/2 |
| quant tree simplification | simplify/2 |
| identify maned objects | transform_special_nps/2 |
| find anaphoric referents | anaphoric_references/3 |

```
IF query
        query evaluation              answer/2
        answer generation             surface_descriptions/2
        answer display                respond/1
ELSE % assertion or rule
        convert to Horn clauses       translate_to_clausal_form/4
        update knowledgebase          update_knowledgebase/2
END
```

A phased approach is avantageous from a software engineering pont of view because it divides the overall task into several smaller, largely independent, tasks. Each of these smaller tasks is easier to comprehend and debug than it would be if the tasks were merged into one large task.

If there is little backtracking between the different phases, one can do more accurate error detection and correction than would be possible in an integrated approach. If there is no backtracking between phases and a particular phase fails, one knows that the error occurred in that phase. One the other hand, if backtracking is allowed between the phases, then failure of a particular phase does not necessarily mean that an error has been found. By backtracking to a choice point in a previous phase, one may obtain a new alternative which is able to pass all the phases. Thus if backtracking is allowed between phases, it is difficult to determine whether failure of a particular phase id due to an input error, or due to a preceding phase not having yet generated the correct alternative.

These are disadvantages to using a phased approach. If one does not allow backtracking between phases, one must ensure that each pahse generates a composite structure which represents all possible alternatives of that phase. Related to this is the fact that an integrated approach may be able to eliminated possibilities faster than a phased approach. In an integrated approach, the constraints of each phase can be applied as soon as they become relevant. However, in a phased approach, the constraints imposed by later phases can not be applied until the analysis has reached those phases.

For example, in an integrated approach, semantic information could be used to avoid building syntactic parse trees for parses which are suntactically valid, but semantically anomalous.

An integrated approach is also more psychologically plausible than a phased approach. It does not seem likely that humans wait until they have heard all the words in a sentence, before they begin to analyse the first few words. Also, response time could be improved if analysis of the first words in a sentence began before the entire sentence had been entered.

Tulip's Knowledge Representation

Tulip uses Prolog as its knowledge representation (KR) language. In fact, with the exception of skolem functions, Tulip's knowledge representation is limited to the Datalog subset of Prolog (Datalog is Prolog without function symbols).

Using Prolog as the knowledge representation has efficiency advantages over susing full first order logic (FOL). Prolog uses the Unique Names Assumption. This means that in Prolog different atoms are assumed to denote different objects. In FOL one must introduce explicit formulae to speficy that different atoms refer to different objects, eg :

tom = jack

Prolog also uses a Closed World Assumption. If a fact is not known to be true, it is assumed to be false.

Prolog is not as expressive as FIL but it implements a subset of FOL which can be efficiently processed.

Skolem Functions

Tulip introduces skolem functions and skolem constants in order to eliminate existential quantifiers. For example, one logical interpretation of the sentence :

every man loves some woman.

is :

for every man, X, there exists a woman, Y, such that X loves Y.

Existential variables can not be represented directly in Prolog. Therefore, a skolem function, eg. skol(45,X), is introduced to take the place of the existential variable Y. The first argument of 'skol'is a number which is used to uniquely identify this particular skolem function. Thus the above statement can be represented in Prolog by:

```
loves(X,skol(45,X))  :-man(X).
woman(skoll(45,X)).
```

Note that the fact that the skolem function depends on X means that each man loves a different woman. The woman that Tom loves is different from the woman Jack loves since skol(45,'Tom') can not be unified with skol(45,'Jack').

Another possible interpretation of:

every man loves a woman.

is :

there exists a woman Y, such that for every man X, X loves Y. ie. every man loves the same woman.

In this case the existential variable would be replaced by a skolem constant, and the Prolog representation would be :

```
loves(X,skol(46))  :-man(X).
woman(skol(46)).
```

Note that in this case the skolem 'function'does not depend on X and is, in fact, a constant.

Binary Predicates

Most facts in Tulip, with the exception of events, are conceptually represented as binary predicates - that is Prolog predicates which take two arguments. [Ferguson 88b] has examples of the binary predicates used by Tulip. In general, the predicate name is the name of a relationship, and the two arguments represente the two objects that have the relationship. It is convenient to express facts in terms of binary relationships because a more complex n-ary relationship can always be expressed as n-1 binary relationships. In a dynamic knowledge base in which the number of attributes associated with a class of object can increase at any time, a binary representation offers more flexibility than an n-ary relationship. In a binary representation, to define a new attribute for an object one just adds a new binary predicate. On the other hand, if an object is represented by an n-tuple, where each argument of the n-tuple represents a particular attribute value, one must replace the n-tuple with an (n+1)-tuple when one adds a new attribute.

A disadvantage of a binary representation with respect to an n-tuple representation is that the binary representation will generally take up more space.

Using binary predicates is essentially equivalent to a semantic net representation. The two arguments of the predicate correspond to two nodes in a semantic net and the binary relationship corresponds to a directed arc between the two nodes. The marc is labelled with the name of the relationship.

Binary predicates can also be conceptualized as an Object-Attribute-Value representation, where the predicate name is the Attribute, the first argument is the Object, and the third argument is the Value.

The actual representation in Tulip is a bit more complex than described above. A binary relationship is represented by a Prolog predicate which takes three arguments - the first argument is the name of the binary relationship and the remaining two arguments are the objects which have that relationship. The predicate name specifies the type of relationship which is being expressed.

For example, the binary relationship expressed in the sentence :

Joe is the father of Tom

would be represented by :

prop0(father,Tom,Joe)

Here, 'father' is the relationship and 'Tom' and 'Joe' are the objects involved in the relationship. The predicate name 'prop0' indicates that this is a 'property'relationship. A property relationship is one which can be expressed as a possessive relationship, eg. Tom's father is Joe. Tulip needs to know the relationship types so that it can generate appropriate responses to queries. For examples, Tulip knows that any property relationship of the form :

prop0(Rel,X,Y)

can be expressed as : X's Rel is Y.

Encoding relationship type information also allows Tulip to answer 'metalevel' queries. For example, the query :

What does Hank have?

is translated into :

```
answer([X]) :-prop(Rel,'Hank', X).
```

where Rel represents a variable relationship. This finds the values of all property relationship which Hank has. If the binary relationships were expressed directly, this query would have the form :

```
answer([X]) :-Rel('Hank', X).
```

which would be illegal in most versions of Prolog because variable predicate names are not allowed.

Know and Inferred Facts

Tulip distinguishes between explicitly know and inferred facts. Explicit facts are expressed by predicates which have a '0'appended to their names. Inferred facts, ie. rules, are represented by predicates which do NOT have a '0'appended to their names. To link up these two predicates a linking clause is used, eg. :

```
prop(Rel,X,Y) :-prop0(Rel,X,Y).
```

Besides these linking clauses, there may also be Prolog rules which establish relationships using inferencing. For example, when generating a description of an object, Tulip only displays the object's direct relationships.

Representation of Verbs

In Tulip, verbs are generally represented by an 'event'predicate. For example, the sentence :

Jack married Jill on Thursday.

is translated (in part) into :

event0(marry,skol(53),'Jack','Jill'),adjunct0(on,skol(53),date(89,3,30))

The first argument of event0 is the citation form of the verb, the second argument is a skolem constant which represents this particular event, the third argument is the subject of the verb, and the fourth argument is the object of the verb. The event skolem constant, skol(53), (the second argument of event0) is used by predicates which express extra information about the event, such as when and where the event occured.

In this case :

adjunct0(on,skol(53),date(89,3,30))

indicates that the event occured on Thursday.

## REFERENCES

[Ferguson 88a] Ferguson, Ronald, Enhancements to TULIP from Jan. 1/88 to Mar. 4/88, 1988.

[Ferguson 88b] Ferguson, Ronald, Translating Words to Their 'Meanings'in Tulip, 1988.

[Ferguson 89] Ferguson, Ronald, Parsing Business Cards with an Extended Logic Grammar, 1989.

[Ferguson 89b] Ferguson, Ronald, Sun TULIP Users Manual, 1989.

[Pereira 83] Pereira, Fernado, Logic for Natural Language Analysis, Technical Note 275, SRI International, 1983.

[Robb 88] Robb, Madeleine, Index of Work Completed by Madeleine Robb on Tulip Lexicons, Aug. 1988.

Gouvernement du Canada
Ministère des Communications

Government of Canada
Department of Communications

**Le Centre canadien de recherche sur l'informatisation du travail**
**Canadian Workplace Automation Research Centre**

Canada

# LE PROJET TULIP

par

**Ron Ferguson**

Ce rapport est le résultat de travaux de recherche effectués au Centre canadien de recherche sur l'informatisation du travail sous la direction de Joël Muzard, Groupe systèmes experts.

Ministère des Communications

Laval

Novembre 1989

Nº de cat. Co 28-1/40-1989F
ISBN 0-662-95734-2

Les opinions émises dans ce rapport n'engagent que l'auteur.

* This report is also available in English

## Table des matières

Introduction.....3

Implantation du système Tulip.....3

Construction du système Tulip.....5

Les fichiers sources du système Tulip.....7

Heuristiques d'analyse de cartes professionnelles.....13

L'architecture du système Tulip.....13

La représentation des connaissances dans le système Tulip.....15

Fonctions de Skolem.....16

Prédicats binaires.....17

Faits connus et inférés.....19

Représentation des verbes.....20

## Introduction

Ce rapport discute de l'implantation du système Tulip dans les postes de travail Sun. Les possibilités offertes par le système Tulip, ainsi que ses limites, sont décrites dans le guide de l'utilisateur du système Sun-Tulip (Ferguson 89b).

## Implantation du système Tulip

La version Sun du système Tulip est écrite en langage Quintus Prolog et comporte quelques sous-programmes C. Elle fonctionne d'après le système d'exploitation UNIX. Le code source est mis à jour à l'aide du système de contrôle de codes sources (SCCS) UNIX. Le code source du système Tulip figure sous le répertoire **/home/condor/ferguson/tulip.** Tous les sous-répertoires mentionnés dans ce rapport sont des sous-répertoires de **/home/condor/ferguson/tulip.** Le répertoire **tulip** contient les sous-répertoires suivants :

| | | |
|---|---|---|
| **main** | - | contient le code source principal pour l'analyse des phrases. |
| **business-card** | - | contient le code pour l'analyse des cartes professionnelles et des noms. |
| **expert-system** | - | contient le code pour les connaissances intégrées de la date et de l'heure, les opérations arithmétiques et les réponses aux questions. |
| **extraposition-grammar** | - | code pour la traduction de grammaires d'extraposition en clauses Prolog. |
| **foreign-files** | - | code de commande pour les interfaces avec les sous-programmes C. |

| | | |
|---|---|---|
| **help** | - | fonction d'assistance du système Tulip. |
| **knowledge** | - | fichiers de connaissances qui peuvent être lus par le système Tulip. |
| **lexicon** | - | dictionnaire du système Tulip. |
| **transform** | - | interpréteur de commandes d'écriture de préprocesseur Prolog. |
| **tulip** | - | images exécutables. |
| **documentation** | - | documentation du système Tulip. |

Construction du système Tulip

Dans le sous-répertoire **main**, les fichiers *parse.pl* et *date-time-parse.pl* contiennent des grammaires d'extraposition [Pereira 83] qui doivent être traduites en clauses Prolog. Les fichiers Prolog correspondants sont *translation-of-parse.pl* et *translation-of-date-time-parse.pl* respectivement. Si l'un des fichiers contenant une grammaire d'extraposition est modifié, il doit être retraduit. Pour retraduire un fichier, chargez Tulip en l'extrayant du sous-répertoire **main** et compilez *xg-fix.pl.* :

compile(.../extraposition-grammar/xg-fix).

Faites ensuite évaluer *gxg* (défini dans *xg-fix.pl*) :

gxg.

Le prédicat *gxg* vous demandera de spécifier des fichiers d'entrée et de sortie. Introduisez *parse.pl* (ou *date-time-parse.pl*) comme fichier d'entrée et *translation-of-parse.pl* (ou *translation-of-date-time-parse.pl*) comme fichier de sortie.

Dans le sous-répertoire **business-card**, le fichier *address.pl* contient une grammaire étendue à clauses définies qui sert à l'analyse des cartes professionnelles [Ferguson 89]. Ce fichier doit être prétraité lorsqu'il est modifié. Pour utiliser le préprocesseur, chargez Tulip comme ci-dessus et faites exécuter *gopt* :

gopt.

Le prédicat gopt vous demandera de spécifier des fichiers d'entrée et de sortie. Introduisez *.../business-card/address.pl* comme fichier d'entrée et *.../business-card/tr-address.pl* comme fichier de sortie.

Le fichier *tulip-system.pl* du sous-répertoire **main** contient les commandes de compilation du système Tulip. Si l'on compile *tulip-system.pl*, le système Tulip est recompilé et un fichier image est sauvegardé dans le sous-répertoire **tulip**. Le sous-répertoire **main** contient un fichier exécutable appelé *tulip* qui charge cette image.

Le système Tulip utilise un lexique très étendu. Celui-ci est constitué de faits Prolog contenus dans une série de fichiers. Lorsque Tulip a été implanté pour la première fois dans des postes de travail Sun, ces fichiers ont été chargés dans le système. Toutefois, le temps de réponse de Tulip sur les machines Sun 3/50 s'est avéré trop long. Il semblerait que cette faiblesse soit en partie causée par la trop grande quantité de mémoire nécessaire pour la pagination, en raison de la taille du programme Tulip. En vue d'accroître la vitesse d'exécution de ce dernier, le lexique a été stocké dans des fichiers de gestion de base de données (dbm) et seules les entrées du lexique qui correspondent aux mots apparaissant dans les phrases d'entrée sont enregistrées en mémoire. En déchargeant ainsi le lexique, on a pu réduire l'espace de mémoire virtuelle nécessaire au programme Tulip, lequel fonctionne maintenant 3,4 fois plus rapidement.

Le lexique de la base de données est constitué d'enregistrements. Chaque enregistrement comprend un champ clé qui est un mot anglais et un champ de contenu qui est un nombre entier. Ce dernier est une représentation vectorielle binaire d'un ensemble qui est un sous-ensemble de «lex_set», lequel est défini dans *lexical-dictionary-interface.pl*. Les éléments de cet ensemble sont les noms des prédicats du lexique Prolog qui ne peuvent avoir pour argument qu'un mot anglais. Par exemple, si *noun(dog))* est un fait dans le lexique Prolog, *noun* est un élément de l'ensemble associé au mot *dog*.

Le lexique de la base de données se trouve dans les fichiers *lexicon.dir* du répertoire **lexicon/dbm/**. Lorsque ces fichiers doivent être reconstitués, vous pouvez suivre la procédure suivante :

Effacez les fichiers *lexicon.pag* et *lexicon.dir*.
Créez des fichiers *lexicon.pag* et *lexicon.dir* vides.
Passez au sous-répertoire **main**.
Compilez *lexicon-system.pl* (ce fichier est un fichier de compilation pour le lexique Prolog).
Chargez le système Tulip sans toutefois entrer (restez dans le système Prolog).
Entrez dans **create_lexicon**.
Prenez note qu'il faut plusieurs heures pour recréer le lexique de la base de données. Si vous ne désirez y apporter que quelques modifications

mineures, il peut s'avérer plus pratique de modifier le lexique existant en utilisant les prédicats des sous-répertoires **dbm.pl** et **fixed-set.pl** qui se trouvent respectivement dans les répertoires **lexicon/dbm** et **main.** Évidemment, en procédant de la sorte, il faudra également effectuer les mêmes modifications dans les fichiers du lexique Prolog. Une liste de ces fichiers est disponible dans le fichier *main/lexicon-system.pl* et une description de ces fichiers est disponible dans [Robb 88].

Certains des fichiers du lexique contiennent des prédicats ayant une arité supérieure à 1. Ces prédicats ne figurent pas dans le lexique de la base de données. Afin de réduire la charge de pagination, le fichier *tulip-system.pl* n'inclut ces fichiers dans le système Tulip que si le fait *lots_of_memory* figure dans la base de données Prolog. Si tel n'est pas le cas, ces fichiers ne sont pas inclus lors de la compilation du système Tulip.

Les fichiers sources du système Tulip

Vous trouverez ci-après une liste des fichiers sources non inclus dans le lexique du système Tulip, ainsi qu'une description de leur contenu. Les noms d'accès de ces fichiers se rapportent au sous-répertoire **main**. Chacun d'eux porte une extension *.pl* qui n'apparaît cependant pas dans la liste.

parse
: Grammaire anglaise principale.

date-time-parse
: Analyse syntaxique de la date et de l'heure.

.../business-card/address
: Analyseur syntaxique de cartes professionnelles.

utilities
: Prédicats utilitaires généraux.

collections
: Opérations sur les ensembles.

...help/help
: Système de menus d'assistance.

tulip-date
: Date avec jour de la semaine.

accept_sentence
: Lecture d'une phrase.

.../transform/transform
: Interpréteur pour transformation de clauses Prolog (préprocesseurs).

tokenize
: Analyseur lexical : chaîne __ liste d'éléments lexicaux (réversible).

lexical_structure
: Définition des types de données des structures lexicales.

guess-lexical-type
: Catégorisation lexicale approximative des mots qui n'apparaissent pas dans le lexique.

.../extraposition-grammar/xg-fix-aux
: Non-terminaux de base pour grammaires d'extraposition et enregistrement de mots analysés.

fp
: Opérations sur les tableaux (listes de listes).

abbrev
: Abréviations.

synonym
: Synonymes.

french-english
: Traduction de mots du français à l'anglais.

new-lexicon
: Lexique de base.

attachment-control
: Masques qui contrôlent le lien modificateur.
Voir F. Pereira, «Logic for Natural Language Analysis», p. 64.

top
: Routines Prolog de haut niveau pour Tulip.

execute_command
: Éxécution des commandes intégrées de Tulip.

chat-lexicon-interface
: Recherche de mots dans le lexique.

morphology
: Analyse morphologique.

slot-filler
: Remplacement des mots par leurs «significations» dans un arbre d'analyse.

simplify
: Simplification des arbres *quant* par élimination des feuilles «vrai».

scope-determination
: Détermination de l'étendue des quantificateurs anglais.

template
: Interface de liaison avec les modèles qui définissent la «signication» des mots.

predicate-calculus
: Conversion en forme clausale des formules de calcul des prédicats.

modify-determiner
: Modification des déterminants des arbres d'analyse d'assertions pour faire en sorte que la quantification soit plus explicitement représentée.

knowledge-file-access
: Lecture et écriture dans les fichiers de connaissances.

transform-answer
: Addition de faits dans la base de connaissances ou consultation de cette dernière.

parse-names
: Analyse d'un nom, d'une adresse ou d'une autre entité non grammaticale.

anaphora
: Traitement des références anaphoriques.

triggers
: Déclenchement d'interventions basées sur les assertions transmises à la base de connaissances.

parse-mods
: Modification de la grammaire pour traiter les conjonctions des syntagmes verbaux.

special-nps
: Analyseurs spéciaux pour identifier les entités non-grammaticales (par exemple, les noms, adresses et numéros de téléphone de personnes et de sociétés).

.../business-card/gr-transform
: Traduction d'une grammaire étendue à clauses définies (pour les cartes professionnelles) en clauses Prolog.

.../business-card/address-util
: Analyse des éléments lexicaux de carte professionnelle.

.../business-card/heuristics
: Heuristiques pour étiqueter les parties d'une carte professionnelle que la grammaire de cartes professionnelles n'était pas en mesure d'analyser.

.../business-card/semantics
: Traitement de la spécification des relations entre les objets de carte professionnelle.

infer
: Inférences de base.

loop-checking
: Vérification des règles pour détecter les boucles récursives qui peuvent s'y trouver.

database-access
: Assertion de clauses révocables.

significant-words
: Extraction d'assertions par association de mots.

date-time-facts
: Faits relatifs à la date et à l'heure.

mrl_temp
: Interface de liaison aux *modèles de représentation de sens* des verbes.

type-hierarchy
: Spécification d'une hiérarchie intégrée pour l'appariement des modèles.

.../expert-system/date-time-expert
: Moteur du système expert s'occcupant de la date et de l'heure, c'est-à-dire, le calculateur.

.../expert-system/talkr
: Évaluation d'une interrogation.

.../expert-system/aggreg
: Éxécution d'opérations arithmétiques.

quintus_kludges
: Prédicats qui ne peuvent être définis correctement (ou qui ne l'ont pas été) dans la version Quintus du système Tulip.

La table précédente a été créée par un programme Prolog du fichier *documentation/file-comments.pl* qui reçoit le commentaire d'en-tête provenant de chaque fichier. Le fichier *main/tulip-system.pl* contient les instructions (commentées) pour faire exécuter le programme de commentaires d'en-tête. Le fichier *main/header-comment.pl* contient un modèle pour les commentaires d'en-tête.

Heuristiques d'analyse de cartes professionnelles

Tel qu'il est exposé dans [Ferguson 89], on a eu recours à des heuristiques pour classifier les parties de cartes professionnelles qui ne pouvaient pas être classifiées par la grammaire de cartes professionnelles de Tulip. Toutefois, cette grammaire peut être appliquée à l'analyse de l'information pour laquelle les heuristiques de cartes professionnelles ne sont pas appropriées (c'est-à-dire les organismes qui oeuvrent dans l'intelligence artificielle). Par conséquent, Tulip a été modifié de façon que les heuristiques ne soient appliqués que lorsque le fait *apply_heuristics* est défini. Le drapeau *apply_heuristics* est vérifié dans le fichier *business-card/address.pl*.

L'architecture du système Tulip

L'architecture du système Tulip est basée sur celle du système Chat-80 [Pereira 83]. Les différences entre les deux systèmes sont discutées dans [Ferguson 88a].

Le système Tulip utilise une architecture phasée pour analyser les phrases. L'analyse de la phrase entière est donc faite suivant plusieurs phases successives. Pour une phrase en anglais, ces phases sont :

| Processus | Prédicat |
|---|---|
| - lecture de l'information d'entrée | read_lines/6 |
| - analyse lexicale | tokenize_parse/3 |
| - consultation du lexique syntaxique | lexical_lookup/2 |
| - localisation des mots significatifs | significant_words/4 |
| - analyse syntaxique | sentence/6 |
| - consultation du lexique sémantique | i_sentence/2 |
| - détermination de l'étendue des poids | clausify/2 |
| - simplication d'un arbre de quantification | simplify/2 |

| | |
|---|---|
| - identification des entités non-grammaticales | transform_special_nps/2 |
| - localisation des références pronominales | anaphoric_references/3 |
| SI requête | |
| évaluation de la requête | answer/2 |
| génération de la réponse | surface_descriptions/2 |
| affichage de la réponse | respond/1 |
| SINON % assertion ou règle | |
| conversion en clauses Horn | translate_to_clausal_ form/4 |
| mise à jour de la base de connaissances | update_knowledgebase/2 |
| FIN | |

L'approche phasée est avantageuse du point de vue du génie logiciel, car elle divise l'ensemble de la tâche en plusieurs tâches plus petites et largement indépendantes. Chacune de ces dernières est plus facile à comprendre et à mettre au point que ne le serait une seule et unique tâche résultant de leur fusion.

S'il y a de petits retours en arrière (backtracking) entre les différentes phases, on peut détecter et corriger les erreurs avec une plus grande précision que dans une approche intégrée. S'il n'y a pas de retour en arrière entre les phases et qu'il y a une défaillance dans l'une des phases, on sait que l'erreur est survenue dans cette phase. Par contre, s'il peut y avoir des retours en arrière entre les phases, on ne peut être sûr d'avoir détecté l'erreur lorsqu'une défaillance survient dans l'une des phases. Par un retour en arrière à un point choisi dans une phase précédente, on pourrait obtenir une nouvelle alternative pouvant passer par toutes les phases. Par conséquent, si l'on autorise les retours en arrière entre les phases, il est difficile de déterminer si une défaillance dans l'une des phases est due à une erreur d'entrée ou au fait qu'une phase précédente n'a pas encore engendré l'alternative correcte.

La méthode phasée a ses inconvénients. S'il n'y a pas de retour en arrière entre les phases, il faut s'assurer que chaque phase engendre une structure composite qui représente toutes les possibilités pour cette phase. De ce fait, l'approche intégrée peut parfois éliminer les possibilités plus rapidement que l'approche phasée. Dans l'approche intégrée, les contraintes de chaque phase peuvent être appliquées à mesure qu'elles deviennent pertinentes. Toutefois, dans l'approche phasée, les contraintes imposées par les phases ultérieures ne peuvent pas être appliquées tant que l'analyse n'a pas atteint ces phases.

Par exemple, dans une approche intégrée, l'information sémantique pourrait être utilisée pour éviter de construire des arbres d'analyse syntaxique pour les analyses syntaxiquement valides, mais sémantiquement anormales.

De plus, l'approche intégrée est plus plausible que l'approche phasée du point de vue psychologique. L'homme n'attend pas d'avoir entendu tous les mots d'une phrase avant de commencer à en analyser les premiers mots. On pourrait améliorer le temps de réponse du système si l'analyse des premiers mots d'une phrase commençait avant que la phrase entière n'ait été introduite dans le système.

<u>La représentation des connaissances dans le système Tulip</u>

Tulip utilise le langage Prolog comme langage de représentation des connaissances. En fait, à l'exception des fonctions de Skolem, la représentation des connaissances dans le système Tulip est limitée au sous-ensemble Datalog de Prolog (Datalog est la langage Prolog sans les symboles de fonction).

Le langage Prolog est parfois plus efficace qu'un langage logique de premier ordre (langage LPO) comme langage de représentation des connaissances. Prolog utilise l'hypothèse de l'unicité des noms. Par conséquent, en Prolog, les atomes différents désignent des objets différents, Dans un langage LPO, il faut introduire des formules explicites pour indiquer que des atomes différents désignent des objets différents, par exemple :

tom ≠ jack

Prolog utilise également l'assomption du monde fermé. Un fait est considéré faux si sa véracité n'a pas été démontrée.
Le langage Prolog n'est pas aussi expressif qu'un langage LPO, mais il constitue un sous-ensemble d'un langage LPO qui peut être traité de façon efficace.

Fonctions de Skolem

Tulip utilise des fonctions et des constantes de Skolem pour éliminer les quantificateurs existentiels. Par exemple, l'une des interprétations logiques de la phrase :

every man loves some woman

est :

Pour chaque homme X, il existe une femme Y telle que X aime Y.

Les variables existentielles ne peuvent être représentées directement en langage Prolog. Par conséquent, une fonction de Skolem, par exemple *skol (45,X)*, est introduite à la place de la variable existentielle Y. Le premier argument de *skol* est un nombre utilisé pour identifier uniquement cette fonction particulière de Skolem. Par exemple, en langage Prolog la phrase ci-dessus peut être représentée par :

```
loves(X,skol(45,X)) :-man(X).
woman(skol(45,X)).
```

À noter : le fait que les fonctions de Skolem dépendent de X signifie que chaque homme aime une femme différente. La femme aimée par Tom est différente de celle aimée par Jack car skol(45,'Tom') ne peut être unifié avec skol(45,'Jack').

La phrase :

every man loves a woman

peut également être interprétée dans le sens suivant :

il existe une femme Y telle que, pour chaque homme X, X aime Y, ou en d'autres termes, chaque homme aime la même femme.

Dans ce cas, les variables existentielles seraient remplacées par une constante de Skolem et la représentation Prolog serait :

```
loves(X,skol(46))  :-man(X).
woman(skol(46)).
```

Ici la fonction de Skolem ne dépend pas de X et est en fait une constante.

Prédicats binaires

Dans le système Tulip, à l'exception des événements, la plupart des faits sont représentés conceptuellement sous forme de prédicats binaires, c'est-à-dire de prédicats à deux arguments (prédicats Prolog). On trouvera dans [Ferguson 88b] des exemples de prédicats binaires utilisés par Tulip. En général, le nom du prédicat est celui d'une relation et les deux arguments représentent les deux objets en relation.

Il est pratique d'exprimer les faits sous forme de relations binaires, car une relation n-aire peut toujours être représentée par n-l relations binaires. Dans une base de connaissances dynamique à l'intérieur de laquelle le nombre d'attributs associés à une classe d'objets peut augmenter en tout temps, une représentation binaire offre plus de souplesse qu'une relation n-aire. Pour définir un nouvel attribut à un objet dans une représentation binaire, il suffit d'ajouter un nouveau prédicat binaire. Dans une représentation n-aire, c'est-à-dire une représentation où chaque argument du n-uplet représente une valeur d'attribut particulière, il faut remplacer le n-uplet par un (n+1)-uplet lorsqu'on ajoute un nouvel attribut.

Comparée à une représentation n-aire, la représentation binaire a l'inconvénient de prendre généralement plus d'espace.

L'utilisation de prédicats binaires équivaut essentiellement à utiliser un réseau sémantique. Les deux arguments du prédicat correspondent à deux noeuds dans un réseau sémantique et la relation binaire correspond à un arc orienté entre ces deux noeuds, l'arc portant le nom de la relation.

Les prédicats binaires peuvent également être conceptualisés par des représentations de la forme objet-attribut-valeur où le nom du prédicat est l'attribut, le premier argument, l'objet, et le troisième argument, la valeur.

La représentation utilisée dans Tulip est un peu plus complexe que celle décrite ci-dessus. Une relation binaire y est représentée par un prédicat Prolog à trois arguments, le premier étant le nom de la relation binaire et les deux autres, les objets en relation. Le nom du prédicat précise le type de la relation exprimée. Par exemple, la relation binaire exprimée par la phrase :

Joe is the father of Tom

serait représentée par

prop0(father,Tom,Joe).

Ici *father* est la relation et *Tom* et *Joe* sont les objets en relation. Le nom du prédicat, *prop0* indique qu'il s'agit d'une relation de propriété, c'est-à-dire une représentation qui peut être exprimée sous la forme d'une relation possessive, par exemple, le père de Tom est Joe. Tulip doit connaître les types de relations pour répondre de façon appropriée aux interrogations. Par exemple, Tulip sait que toute relation de propriété de la forme :

prop0(Rel,X,Y)

peut être exprimée comme suit :

X's Rel is Y.

Grâce au codage des types de relations, Tulip peut répondre aux interrogations de méta-niveau.

Par exemple, l'interrogation :

What does Hank have?

devient :

answer([X]) :-prop(Rel,'Hank', X)

où Rel représente une relation variable. Ceci permet de trouver les valeurs de toutes les relations de propriété de Hank. Si les relations binaires étaient exprimées directement, cette interrogation prendrait la forme :

answer([X]) :-Rel('Hank', X)

qui serait interdite dans la plupart des versions de Prolog parce que les noms de prédicats variables ne sont pas autorisés.

Faits connus et inférés

Le système Tulip fait la distinction entre les faits connus explicitement et les faits inférés. Les faits explicites sont exprimés par des prédicats dont le nom porte l'extension *0*. Les faits inférés, c'est-à-dire les règles, sont représentés par des prédicats dont le nom ne porte PAS l'extension *0*. Pour relier ces deux prédicats, on utilise une clause de liaison, par exemple :

prop(Rel,X,Y) :-prop0(Rel,X,Y).

En plus de ces clauses de liaison, il peut également y avoir des règles Prolog établissant des relations par inférence. Par exemple, le fait que

Y is the wife of X if X is the husband of Y

peut être exprimé par la règle Prolog : prop(wife,X,Y) :

-prop(husband,Y,X).

Il est utile de pouvoir distinguer les relations directes (celles dont le nom de prédicat porte l'extension *0*) des relations inférées.

Par exemple, lorsque Tulip décrit un objet, il n'affiche que les relations directes de ce dernier.

Représentation des verbes

Dans le système Tulip, les verbes sont généralement représentés par un prédicat *event*.

Par exemple, la phrase :

Jack married Jill on Thursday

est traduite (en partie) par :

event0(marry,skol(53),'Jack','Jill'),adjuct0(on,skol(53),date(89,3,30)).

Le premier argument de *event0* est la forme citée du verbe, le deuxième, une constante de Skolem représentant l'événement en question, le troisième, le sujet du verbe, et le quatrième l'objet du verbe. La constante de Skolem de l'événement, skol(53) (le deuxième argument de event0), est utilisée par les prédicats qui donnent des informations supplémentaires relatives à l'événement, telles que l'instant et le lieu.

Dans le cas présent,

adjunct0(on,skol(53),date(89,3,30))

indique que l'événement est survenu jeudi.

## RÉFÉRENCES

[Ferguson 88a]
FERGUSON, Ronald. Enhancements to TULIP from Jan. 1/88 to mar. 4/88, 1988.

[Ferguson 88b]
FERGUSON, Ronald. Translating Words to Their *Meanings* in Tulip, 1988.

[Ferguson 89]
FERGUSON, Ronald. Parsing Business Cards with an Extented Logic Grammar, 1989.

[Ferguson 89]
FERGUSON, Ronald. Guide du l'utilisateur du système Sun-Tulip, 1989.

[Pereira 83]
PEREIRA, Fernado. Logic for Natural Language Analysis, Technical Note 275, SIR International, 1983.

[Robb 88]
ROBB, Madeleine. Index of Work Completed by Madeleine Robb on Tulip Lexicons, Aoust 1988.